On the relations between dynamical systems and boolean circuits.

Pascal Koiran

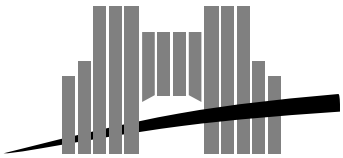
Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
Institut IMAG
Unité de recherche associée au CNRS n°1398

On the relations between dynamical systems and boolean circuits

Pascal Koiran

Janvier 93

N^{O} 93-01



Ecole Normale Supérieure de Lyon
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

On the relations between dynamical systems and boolean circuits

Pascal Koiran

Abstract

We study the computational capabilities of dynamical systems defined by iterated functions on $[0,1]^n$. The computations are performed with infinite precision on arbitrary real numbers, like in the model of analog computation recently proposed by Hava Siegelmann and Eduardo Sontag. We concentrate mainly on the low-dimensional case and on the relations with the Blum-Shub-Smale model of computation over the real numbers.

Keywords: analog computation, non-uniform complexity, dynamical systems.

Résumé

Nous étudions la puissance de calcul de systèmes dynamiques définis par des itérations de fonctions sur $[0,1]^n$. Les calculs sont effectués en précision infinie sur des nombres réels quelquonques, comme dans le modèle de calcul analogique récemment proposé par Hava Siegelmann et Eduardo Sontag. Nous insistons surtout sur le cas des systèmes en petites dimensions et sur les relations avec le modèle de calcul sur les nombres réels de Blum, Shub & Smale.

Mots-clés : calcul analogique, complexité non uniforme, systèmes dynamiques.

On the relations between dynamical systems and boolean circuits*

Pascal Koiran
LIP – IMAG
Unité de Recherche Associée 1398 du CNRS
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cedex 07
France
e-mail: koiran@lip.ens-lyon.fr

January 4, 1993

1 Introduction

A new model of analog computation was recently proposed by Siegelmann & Sontag [7]. In this model, computations are performed by recurrent neural networks with real weights, instead of rational weights in a previous work [8]. It is assumed that the computations are carried out with unbounded precision. Siegelmann & Sontag showed that arbitrary (even non-computable) functions can be computed in exponential time by their networks, and that the class of polynomial-time computable functions is the class P/poly of functions computable by polynomial-size boolean circuits. They also proved that using much more general networks (called *generalized processor networks*) does not result in any gain in computational capabilities.

In this paper, we continue the study of the computational capabilities of low-dimensional dynamical systems started in [3], and generalize some of the results of [7]. Our framework is very similar to Siegelmann & Sontag's: the dynamical systems considered are iterated functions on $[0,1]^n$; some of

*This work was partially supported by the Programme de Recherches Coordonnées C^3 of the CNRS and the Ministère de la Recherche et de la Technologie.

these functions can be described by a finite set of parameters which can take arbitrary (even non-computable) real values.

It is shown in section 3 that iterated piecewise-linear functions on the interval can compute arbitrary functions in exponential time. This is an interesting difference with the "rational weights" case, for which it seems that two dimensions are necessary in order to compute arbitrary recursive functions with iterated piecewise-linear functions [3]. In dimension two, the class of polynomial-time computable functions is exactly P/poly. We then show that using other types of functions or higher dimensions does not allow to compute anything else than P/poly in polynomial time. This result precisely holds for the class of iterated *polynomial-time approximable functions*, which should include most functions of practical interest (it does include piecewise-linear functions, as shown in section 2, as well as multivariate polynomials [7]).

In section 4, we consider on-line systems in which the input is fed to the system bit by bit, instead of being encoded entirely in the initial state. This is closer to the mode of operation of Siegelmann & Sontag's recurrent networks, and it can be shown that these systems are actually equivalent to their generalized processor networks. However, we believe that our model is simpler and more natural. It is shown that on-line systems are in fact a special case of the off-line systems of section 3. Hence the same limitation applies to their computational capabilities.

Finally, these models are compared to the BSS model of analog computation, recently proposed by Blum, Shub and Smale [2]. It is shown that their model is at least as powerful as our's. Whether it is actually more powerful is an interesting open problem, which could shed light on the P = NP problem in the BSS model.

2 Preliminaries

The definitions and notations used throughout the paper are listed below.

I is the unit interval $[0,1]$. $e_i(x)$ is the i^{th} component of $x \in \boldsymbol{R}^d$. For $x, y \in \boldsymbol{R}^d$, $|x-y| = ||x-y||_\infty = \sup_{1 \le i \le d} |x_i - y_i|$.

L^+ is the set of finite non-empty words on the alphabet L.

We identify a natural number with its radix-2 expansion, so that $\boldsymbol{N}$ can be identified with $\{0,1\}^+$. $|x|$ is the length of $x \in \{0,1\}^*$ or $x \in \boldsymbol{N}$ (and not the absolute value).

D is the set of "decimal" numbers in radix 2: $x \in D_n$ iff $x = 0.x_1 \ldots x_n$

with $x_i \in \{0,1\}$, and $D = \cup_{n\geq 1} D_n$.

We say that the digits $x_1 \ldots x_n$ ($x_i \in \{0,1\}$) are the first n digits of $x \in I$ if $y = 0.x_1 \ldots x_n$ is such that $|x - y| \leq 2^{-n}$ (there is of course no uniqueness). We note $y = \mathrm{Trunc}_n(x)$. Similarly, we define $\mathrm{Trunc}_n(x)$ for $x \in I^d$ by taking the first n digits in each component.

Definition 1 *PL_d is the set of piecewise-linear continuous functions on I^d. More precisely, $f : I^d \to I^d$ belongs to PL_d if*

- *f is continuous;*
- *there is a finite set $\mathcal{P}$ of convex closed polyhedra (of non-empty interior) such that f is affine on each $P \in \mathcal{P}$, $I^d \subset \bigcup_{P\in\mathcal{P}} P$ and $\overset{\circ}{P} \cap \overset{\circ}{Q} = \emptyset$ for $P, Q \in \mathcal{P}$, $P \neq Q$.*

Let us recall the classical notion of polynomial-time computable real function [6].

Definition 2 *$f : I^d \to I^d$ is in P_d if there is a Turing machine $\mathcal{T}$ such that*

1. *$\mathcal{T}$ receives as input an integer p specifying the precision required for the output $f(x)$.*
2. *$\mathcal{T}$ computes the number m of input digits needed.*
3. *$\mathcal{T}$ queries an oracle to obtain $\mathrm{Trunc}_m(x)$ (this step takes constant time by definition).*
4. *$\mathcal{T}$ then computes the output $\mathrm{Trunc}_p(f(x))$.*
5. *There are constants a and k such that for any $x \in I^d$ and $p \in \boldsymbol{N}$, the whole computation is performed in time at most ap^k.*

Note that we can assume without loss of generality that m is polynomial in p, since a Turing machine cannot read more than a polynomial number of bits in polynomial time. For encoding functions, we shall use the following notion of polynomial-time computability.

Definition 3 *A function $\phi : \{0,1\}^+ \to I^d$ is in PE_d if there is a Turing machine $\mathcal{T}$ such that*

1. *$\mathcal{T}$ receives as input $x \in \{0,1\}^+$ and an integer p specifying the precision required for the output $\phi(x)$.*

2. *$\mathcal{T}$ computes the output* $\mathrm{Trunc}_p(\phi(x))$.

3. *There are constants a and k such that for any $x \in I^d$ and $p \in \boldsymbol{N}$, the whole computation is performed in time at most $a(p + |x|)^k$.*

P_d and PE_d are in fact subsets of the class of polynomial-time computable functions from $\boldsymbol{R}^m$ to $\boldsymbol{R}^n$ [6].

The transition functions of our systems cannot be allowed to be arbitrary polynomial-time computable functions (in order to show that circuit families can simulate iterated functions, a Lipschitz property is necessary). We shall consider only the following subclass.

Definition 4 *$f : I^d \to I^d$ is in LP_d if $f \in P_d$ and f is Lipschitz (i.e., if there is a constant C such that $\forall x, y \in I^d, |f(x) - f(y)| \leq C|x - y|$).*

It can be easily seen that $f \in \mathrm{LP}_d$ iff it is polynomial-time approximable in the sense of Siegelmann & Sontag [7]:

Lemma 1 *$f : I^d \to I^d$ is in LP_d iff*

- *f is Lipschitz.*
- *There is a Turing machine $\mathcal{T}$ that can compute $\mathrm{Trunc}_n(f(x))$ in polynomial time on an input $x \in D_n^d$.*

Proof. Assume that f is polynomial-time approximable. Given $x \in I^d$ and $n \in \boldsymbol{N}$, we need to compute y such that $|y - f(x)| \leq 2^{-n}$. This can be done by taking $y = \mathrm{Trunc}_p(f(\mathrm{Trunc}_p(x))$, for p large enough. The computation time is polynomial in p by hypothesis. Since f is Lipschitz,

$$\begin{aligned} |y - f(x)| &\leq |\mathrm{Trunc}_p(f(\mathrm{Trunc}_p(x)) - f(\mathrm{Trunc}_p(x))| + |f(\mathrm{Trunc}_p(x)) - f(x)| \\ &\leq 2^{-p} + 2^{k-p} \end{aligned}$$

where 2^k ($k \in \boldsymbol{N}$) bounds the Lipschitz constant of f. Hence $p = k + n + 1$ is sufficient.

The converse is obvious. □

In order to deal with arbitrary real numbers, we need to consider the class $\mathcal{C}/\mathrm{poly}$ for each class $\mathcal{C}$ defined above. The functions in these classes are computed by Turing machines using polynomial advice funtions [1]. Contrary to oracles, the value of the advice function does not depend fully on the input of the Turing machine, but only on its length. For instance, $f \in PE_d/\mathrm{poly}$

if there is a function $A : \boldsymbol{N} \to \{0,1\}^+$ (the advice function) and a Turing machine $\mathcal{T}$ such that on the input $(p, x, A(p + |x|))$, $\mathcal{T}$ computes the first p digits of $f(x)$ in time $O((p + |x|)^k)$. The length of the advice sequence $A(p + |x|)$ must also be polynomial in $p + |x|$ (this latter requirement is in fact redundant, since the portion of the tape that can be scanned by $\mathcal{T}$ in polynomial time is of polynomial size).

If $\mathcal{C} = \mathrm{P}$ is the class of sets $E \subset \{0,1\}^+$ recognizable in polynomial time by Turing machines, we obtain the class P/poly. It will be shown that this is exactly the class of sets that can be recognized in polynomial time by iterated functions. A useful characterization of P/poly is the following. $f : \{0,1\}^+ \to \{0,1\}$ is in P/poly if and only if it can be computed by boolean circuits of polynomial size [1], i.e., if there is a family of circuits $(C_n)_{n \geq 1}$ such that the size of C_n is polynomial in n, and C_n computes $f_{|\{0,1\}^n}$.

It can be easily seen that there are non-computable functions is P/poly, since the circuit family $(C_n)_{n \in \boldsymbol{N}}$ is not supposed to be recursive. Conversely, some computable functions are not in P/poly, since EXPTIME$\not\subset$P/poly [1].

The next result is important because all the specific examples of systems considered in this paper as well as in [7] are piecewise-linear.

Theorem 1 *$PL_d \subset LP_d$/poly.*

Proof. $f \in PL_d$ is clearly Lipschitz. Inside a given polyhedron $P \in \mathcal{P}$, $e_j(f(x)) = \sum_{i=1}^{d} w_{ij} x_i + \theta_j$. The Lipschitz constant of $f_{|P}$ is $L_P = \max_j \sum_{i=1}^{d} |w_{ij}|$. The Lipschitz constant of f is $\max_{P \in \mathcal{P}} L_P$.

In order to show that $f \in \mathrm{P}_d$, we first show that f is a Lipschitz function of its parameters. We will now write $f(W, x)$ in order to take them into account explicitly. Let us first define a convention allowing to associate a unique vector of parameters $W \in \boldsymbol{R}^k$ to f. We can assume that $\mathcal{P}$ is a triangulation, i.e., each polygon P is a polytope (the convex hull of $d + 1$ points in general position). f is uniquely defined by the list W of the vertices of the polytopes, together with the value of f at these vertices, enumerated in a fixed order. We also assume without loss of generality that I^d is in the interior of $\cup_{P \in \mathcal{P}} P$, so that f is defined, continuous and piecewise-linear in a neighborhood of I^d. Some vectors $W \subset \boldsymbol{R}^k$ are not associated to a triangulation of I^d (this occurs when two polytopes intersect); others are associated to a triangulation that does not cover I^d. These vectors are called *invalid*. The set of valid vectors is open: for any valid vector W_0, there is a parallepiped $B(W_0, r)$ of valid vectors. We will now work inside

such a parallepiped, and consider for a given $x \in I^d$ the function

$$\begin{array}{rcl} g & : & B(W_0, r) \longrightarrow \boldsymbol{R}^d \\ & & W \mapsto f(W, x) \end{array}$$

This function is continuous, and piecewise-linear since the equations determining the polytope in which x is located are linear. Hence the analysis of $x \mapsto f(W, x)$ made at the beginning of the proof can be applied to g: g is Lipschitz, and its Lipschitz constant is smaller than $\sum_{i=1}^{d} x_i \leq d$.

We are now ready to show how to compute $f(W_0, x)$ in polynomial time. According to lemma 1, we can assume that $x \in D^d$. An approximation of $f(W_0, x)$ can be obtained simply by computing $f(\mathrm{Trunc}_q(W_0), x)$ for q large enough (this makes sense because the set of valid vectors is open). Since g is Lipschitz, $q = n + C$ is sufficient to obtain n digits of $f(W_0, x)$, for some constant C. The computation clearly takes polynomial time, and the advice sequence is $\mathrm{Trunc}_q(W_0)$. $\square$

3 Off-line inputs

In this section we study a mode of computation in which the input to the system is encoded entirely in its initial state.

Definition 5 *The decision function $F : \{0,1\}^+ \to \{0,1\}$ is off-line computable if it can be computed by an off-line system $\mathcal{S} = (f, \phi, [a_0, b_0], [a_1, b_1])$. $\phi : \{0,1\}^+ \to I^d$ is an encoding function, $[a_0, b_0]$ and $[a_1, b_1]$ ($a_1 > b_0$) are two disjoint decision intervals, and f is a function on I^d.*

On an input $u \in \{0,1\}^+$, the computation performed by $\mathcal{S}$ is defined by the following sequence $(x(t))_{t \in \boldsymbol{N}}$:

- $x(0) = \phi(u)$;
- $x(t+1) = f(x(t))$ *for* $t \geq 0$.

Let $t(u) = \min\{t; x_1(t) \in [a_0, b_0] \cup [a_1, b_1]\}$ (it is assumed that $t(u)$ exists). The output is $F(u) = 0$ if $x_1(t(u)) \in [a_0, b_0]$, and $F(u) = 1$ otherwise.

The time complexity of this computation is the function T such that $T(n) = \max_{|u|=n} t(u)$.

Theorem 2 *An arbitrary total function $F : \{0,1\}^+ \to \{0,1\}$ can be off-line computed by $f \in PL_1$ in exponential time. Moreover, the encoding function is in PE_1, and is independent of F. The decision intervals are also independent of F and are such that $a_0 = b_0 \in \boldsymbol{Q}$ and $a_1 = b_1 \in \boldsymbol{Q}$.*

Proof. It is clearly possible to replace I by the interval $[0,2]$. We shall work with radix-4 encodings, using the digits 1 and 3 only (these encodings were introduced in [8] and used subsequently in [7] and [3]). Hence it will be convenient to consider the function G such that $G(N) = 1$ if $F(N) = 0$, $G(N) = 3$ if $F(N) = 1$, instead of F itself. Contrary to the convention used in the rest of the paper, all decimal expansions in this proof are radix 4.

Let $\phi(N) = 1.1(10)^N 3^{+\infty}$ (N is now viewed as an integer rather than a word on the alphabet $\{0,1\}$). On $[1.1,2]$, set

$$f(x) = 2^{-(p+1)}[2(x - 1.1) + a] + q_0,$$

with $a = 0.0G(0)0G(1)\ldots0G(n)0\ldots$ and $q_0 = 0.q_{01}\ldots q_{0p}$ the initial state of a pushdown automaton $\mathcal{A}$ described below. p is such that $\mathcal{A}$ has less than 2^p states. Therefore

$$f(\phi(N)) = 0.q_{01}\ldots q_{0p}1G(0)1G(1)\ldots1G(N-1)3G(N)3G(N+1)3\ldots.$$

The job of $\mathcal{A}$ is just to pop the sequence of digits $1G(0)1G(1)\ldots1G(N-1)3G(N)3G(N+1)3\ldots$ viewed as a downward infinite stack. This can be done by a function of PL_1 as explained in [3]. When $\mathcal{A}$ reads a 3 at an odd position in this sequence, it knows that the next digit is $G(N)$. After reading $G(N)$, it erases the stack, and enters one of the two halting states a_0, a_1. Finally, the "gap" $[1, 1.1]$ can be filled by an affine function, so as to obtain a continuous piecewise-linear function on $[0,2]$.

In order to compute the n^{th} first digit of $\phi(N)$, the main task is to compare n and $2N+2$, which can be done in time polynomial in $\log n$ and $\log N$. Hence ϕ is in PE_1. □

A comparison with "rational weights" systems [8, 3] is in order. If the parameters defining $f \in PL_d$ are rational, the iterations of f can be computed exactly on a conventional computer (assuming that the initial state is rational); hence it is only possible to compute recursive functions in this model. It turns out that all recursive functions can be computed [8], but two dimensions seem to be necessary [3] (we have proved this using a somewhat restrictive definition of what it means to to compute a recursive function). Theorem 2 shows that one dimension suffices in the "real weights" model. But according to theorem 3, two dimensions are (presumably) better if one is concerned with efficiency issues.

Theorem 3 *Let $C = (C_n)_{n\in\mathbf{N}}$ be a circuit family such that the size $S(n)$ of C_n is a non-decreasing function of n. C can be simulated in time $O(nS(n)^k)$*

(for some fixed $k > 0$) by an iterated function $f \in PL_2$. Moreover, the encoding function ϕ is in PE_2*, and is independent of C. The decision intervals are also independent of C and are such that $a_0 = b_0 \in \boldsymbol{Q}$ and $a_1 = b_1 \in \boldsymbol{Q}$.*

Proof. We follow Siegelmann & Sontag's method [7], and work with $[0,2]^2$ like in the previous proof. It will be convenient to consider that F is defined on $\{1,3\}^+$ instead of $\{0,1\}^+$. Contrary to the convention used in the rest of the paper, all decimal expansions in this proof are to the radix 4. The encoding function is

$$\phi(x) = \phi(x_1 \ldots x_n) = (0.x_1 \ldots x_n, 1.1).$$

From this point, we make a transition to

$$f(\phi(x)) = (0.q_{01} \ldots q_{0p} x_1 \ldots x_n, c),$$

where $q_{o1} \ldots q_{0p}$ encodes the initial state of a Turing machine $\mathcal{M}$ to be defined below, and the digits of $c \in [0,1]$ encode the sequence $C = (C_n)_{n \in \boldsymbol{N}}$. From [3], we know that there will be a function of PL_2 simulating $\mathcal{M}$ in real time. With the initial tape xc, The job of $\mathcal{M}$ is simply to run C_n on the input $x_1 \ldots x_n$. This is an instance of the Circuit Value Problem, which is known to be in P [1]. The overall computation time is $O(nS(n)^k)$ since the encoding of C_n is at a distance $O(nS(n)\log S(n))$ of the initial read-write head (a circuit of size S can be encoded in space $O(S \log S)$). □

Corollary 1 *A function $F : \{0,1\}^+ \to \{0,1\}$ in* P/poly *can be off-line computed in polynomial time by $f \in PL_2$ with an encoding function in* PE_2*.*

Proof. A function in P/poly has polynomial size circuits [1]. □

Since an arbitrary function has exponential size circuits (e.g., its disjunctive normal form for each input size), we could add to this corollary that arbitrary functions can be computed in exponential time, but this is already known from theorem 2.

Theorem 4 *If $F : \{0,1\}^+ \to \{0,1\}$ can be off-line computed in polynomial time by $f \in LP_d/poly$ with an encoding function $\phi \in PE_d/poly$, then $F \in$* P/poly*.*

This result shows that theorems 2 and 3 are in a sense optimal: polynomial time computation in higher dimensions is not more powerful than in dimension two, and the exponential computation time of theorem 2 cannot be

reduced to polynomial time, even in higher dimensions. It is also useless to work with classes of functions more powerful than PL_d. Finally, note that theorem 4 deals with encoding functions in $\mathrm{PE}_d/\mathrm{poly}$, although PE_d is all what is needed for theorems 2 and 3. The reason is that encoding functions in $\mathrm{PE}_d/\mathrm{poly}$ will be useful in the next section.

The proof of theorem 4 is similar to Siegelmann and Sontag's analysis of generalized processor networks. We first need the following result.

Lemma 2 *Let $(f_t)_{t\in\mathbf{N}}$ be a family of Lipschitz functions on I^d having their Lipschitz constants uniformly bounded above by $C > 1$, and $\epsilon > 0$. Let $(x(t))_{t\in\mathbf{N}}$ and $(\overline{x}(t))_{t\in\mathbf{N}}$ be two sequences defined by $x(t+1) = f_t(x(t))$ and*

$$\overline{x}(t+1) = f_t(\overline{x}(t)) + \epsilon(t)$$

with $\forall t \geq 0,\ |\epsilon(t)| \leq \epsilon$ and $|\overline{x}(0) - x(0)| \leq \epsilon$. Then

$$\forall t \geq 0,\ |\overline{x}(t) - x(t)| \leq \frac{\epsilon}{C-1}(C^{t+1} - 1).$$

Proof. By induction. The result is true for $t = 0$. Assume that it holds at time t. Then

$$\begin{aligned} |\overline{x}(t+1) - x(t+1)| &\leq C|\overline{x}(t) - x(t)| + \epsilon \\ &\leq \frac{C\epsilon}{C-1}(C^{t+1} - 1) + \epsilon \quad = \quad \frac{\epsilon}{C-1}(C^{t+2} - 1). \end{aligned}$$

□

In the next section, this lemma will be applied to a family of functions, but it will be now used with $f_t = f$ for all t.

Proof of theorem 4. In order to distinguish between output 0 (interval $[a_0, b_0]$) and output 1 (interval $[a_1, b_1]$), let us compute each vector $x(t)$ with the accuracy $(a_1 - b_0)/3$ for $0 \leq t \leq t(u)$ (recall that $t(u)$ is the computation time on input u, and that $n = |u|$). Starting from $\overline{x}(0) = \mathrm{Trunc}_q(\phi(u))$, define $\overline{x}(t+1) = \mathrm{Trunc}_q(f(\overline{x}(t))$. Let $C > 1$ be an upper bound of the Lipschitz constant of f. According to lemma 2, it is sufficient to have

$$\epsilon = 2^{-q} \leq \frac{(a_1 - b_0)(C-1)}{3(C^{T(n)} - 1)}.$$

Hence we can take $q = O(T(n)) = O(n^k)$, where k is a constant. Since $f \in \mathrm{P}_d$, $\overline{x}(t+1)$ can be computed from $\overline{x}(t)$ in polynomial time with a polynomial advice sequence. Since $\phi \in \mathrm{PE}_d$, $\overline{x}(0)$ can be computed in time polynomial in $q + n$ with an advice sequence polynomial in $q + n$, hence polynomial in n. □

4 On-line inputs

In this section, we study a mode of computation in which the input is fed to the system bit by bit starting from a fixed initial state, instead of being encoded entirely in the initial state. It can be shown that these on-line systems are equivalent to Siegelmann & Sontag's generalized processor networks. We will not use their model, because we believe that our's is simpler and more natural.

Definition 6 *The decision function* $F : \{0,1\}^+ \to \{0,1\}$ *is on-line computable if it can be computed by an on-line system* $\mathcal{S} = (f, f_0, f_1, [a_0, b_0], [a_1, b_1])$. $f_0, f_1 : I^d \to I^d$ *are encoding functions,* $[a_0, b_0]$ *and* $[a_1, b_1]$ *(*$a_1 > b_0$*) are two disjoint decision intervals, and* f *is a function on* I^d.

On an input $u \in \{0,1\}^+$, *the computation performed by* $\mathcal{S}$ *is defined by the following sequence* $(x(t))_{t \in \mathbf{N}}$:

- $x(0) = 0$;
- $x(t+1) = f_{u_{t+1}}(x(t))$ *for* $0 \leq t \leq n-1$.
- $x(t+1) = f(x(t))$ *for* $t \geq n$.

Let $t(u) = \min\{t; x_1(t) \in [a_0, b_0] \cup [a_1, b_1]\}$ *(it is assumed that* $t(u)$ *exists). The output is* $F(u) = 0$ *if* $x_1(t(u)) \in [a_0, b_0]$, *and* $F(u) = 1$ *otherwise.*

The time complexity of this computation is the function T *such that* $T(n) = \max_{|u|=n} t(u)$.

Slightly different definitions of on-line systems are conceivable. One might for instance consider that a system is defined by only two functions f_0 and f_1, and that the input is an infinite word $u \in \{0,1\}^\omega$ with $u_i = 0$ for i large enough (this is closer to Siegelmann & Sontag's neural networks).

On-line systems are clearly a special case of the off-line systems discussed in section 3, with the encoding function

$$\phi(u_1 \ldots u_n) = f_{u_n} \circ \ldots f_{u_2} \circ f_{u_1}(0). \tag{1}$$

Theorem 5 *If* $f_0, f_1 \in \mathrm{LP}_d/\mathrm{poly}$, *the encoding function of equation (1) is in* $\mathrm{PE}_d/\mathrm{poly}$.

Proof. We show how to compute $\phi(u)$ with accuracy 2^{-p} in time $O((p+n)^k)$, for $u = u_1 \dots u_n \in \{0,1\}^+$. Starting from $\overline{x}(0) = 0$, compute $\overline{x}(t+1) = \mathrm{Trunc}_q(f_{u_t}(\overline{x}(t))$, for q large enough and $0 \leq t \leq n-1$. According to lemma 2, it is sufficient to have

$$\epsilon = 2^{-q} \leq \frac{2^{-p}(C-1)}{C^{n+1}-1},$$

where C bounds the Lipschtz constants of f_0 and f_1 (we can assume that $C \geq 2$ without loss of generality). Hence we can take $q = \lceil p + (n+1)\log C \rceil$. Since f_0 and f_1 are in $\mathrm{P}_d/\mathrm{poly}$, $\overline{x}(t+1)$ can be computed from $\overline{x}(t)$ in polynomial time using an advice sequence polynomial in q.

The computation time is clearly polynomial in the input length $s = n+p$, with a polynomial advice sequence. However, this sequence must depend on s only. In order to insure this, just replace the value of q used up to now by the larger value $q = \lceil \log C \rceil (s+1)$. □

It follows immediately that on-line systems are not more powerful than off-line systems.

Corollary 2 *If $F : \{0,1\}^+ \to \{0,1\}$ is on-line computable in polynomial time, $F \in \mathrm{P}/\mathrm{poly}$.*

Proof. Straightforward consequence of theorems 4 and 5. □

The converse can also easily be proven: on-line systems are thus equivalent to off-line systems with respect to polynomial-time computations.

Theorem 6 *An arbitrary function $F \in \mathrm{P}/\mathrm{poly}$ can be computed in polynomial time by an on-line system with f_0, f_1 and f in PL_2.*

Proof. Recall that the encoding function of theorem 3 is

$$\phi(x) = \phi(x_1 \dots x_n) = (0.y_1 \dots y_n, 1.1),$$

with $x_i \in \{0,1\}$ and $y_i = 2x_i + 1 \in \{1,3\}$. Here we will take $\phi(x) = (0.y_n \dots y_1, 1.1)$ (this is equivalent, since the content of a stack of a two-stack pushdown automaton can be reversed in linear time. It is then possible to perform the computation using the same function f as in theorem 3). The two encoding functions below are clearly suitable:

- $f_0(z_1, z_2) = (\dfrac{1+z_1}{4}, 1.1)$;

- $f_1(z_1, z_2) = (\frac{3 + z_1}{4}, 1.1)$.

□

Both types of systems are also equivalent with respect to exponential time computations.

Theorem 7 *An arbitrary function $F : \{0,1\}^+ \to \{0,1\}$ can be computed by an on-line system (f_0, f_1, f) such that $f \in PL_1$ and $f_0, f_1 : \boldsymbol{R} \to \boldsymbol{R}$ are polynomials with rational coefficients.*

Proof. Recall that the encoding function of theorem 2 is

$$\phi(N) = \phi(u_1 \dots u_n) = 1.1(10)^N 3^{+\infty}.$$

Here we use the encoding $\psi(N) = \phi(0) - \phi(N)$, because the initial state of an on-line system is always 0. Once $\psi(N)$ is computed, it is clearly possible to compute $\phi(N) = \phi(0) - \psi(N)$ and to apply the same method as in the proof of theorem 2, by iterating a function $f \in PL_1$.

We need to find f_0 and f_1 such that

$$f_{u_k} \circ \dots \circ f_{u_1}(0) = \psi(N_k)$$

with $N_k = u_1 \dots u_k$. There are non-zero rational constants a and b such that $\psi(N) = a(1 - b^N)$. Since $N_{k+1} = 2N_k + u_{k+1}$, $\psi(N_{k+1}) = a\left[1 - b^{u_{k+1}}(b^{N_k})^2\right]$. $b^{N_k} = 1 - \psi(N_k)/a$, hence $f_i(x) = a\left[1 - b^i(1 - x/a)^2\right]$. □

5 The Blum, Shub & Smale model

This is a model of analog computation based on real RAM machines similar to the standard RAM machines studied in discrete complexity theory. The main difference between discrete and real RAMs is that in the latter model a register can store an entire real number (with infinite precision), instead of an integer in the discrete case (see [2] and [4] for more details). The basic arithmetic and logical operations ($+$,$-$,$\times$,$/$,$\leq$) can be performed in constant time.

The purpose of the Blum, Shub & Smale (BSS) model is to provide a formal model of computing on real-valued data. Nevertheless, we are free to consider the restriction of this model to discrete inputs and outputs, in

order to make a comparison with the models studied in the rest of the paper. Theorem 8 shows that the BSS model is at least as powerful.

A few notations: P_R is the set of functions computable in polynomial time in the BSS model; P_{RD} is the restriction of P_R to discrete inputs and outputs (i.e., $P_{RD} = P_R \cap \{f : \{0,1\}^+ \to \{0,1\}\}$). For non-deterministic machines, the classes NP_R and NP_{RD} are defined similarly.

Theorem 8 *In the BSS model, an arbitrary function $F : \{0,1\}^+ \to \{0,1\}$ can be computed in exponential time. For polynomial-time computation,* $\mathrm{P/poly} \subset \mathrm{P}_{RD}$ *and* $\mathrm{NP/poly} \subset \mathrm{NP}_{RD}$.

Proof. We adapt the ideas of the proofs of theorems 2 and 3.

In order to compute $F : \{0,1\}^+ \to \{0,1\}$, we consider again the real number $c = 0.F(0)F(1)\ldots F(N)\ldots$. Since $F(N) = \lfloor 2^{N+1}c \rfloor$, $F(N)$ can be computed by shifting c $(N+1)$ times to the left (multiplication by 2) and subtracting the leftmost digit (subtraction and comparison). This clearly takes exponential time in $\log N$.

In order to compute $F \in \mathrm{P/poly}$, we simulate a two-stack pushdown automaton, using one register for each stack. This can clearly be done like in theorem 3.

In order to compute $F \in \mathrm{NP/poly}$, we use directly the definition of NP/poly in terms of advice sequences (this could also have been done in the proof of corollary 1; conversely, non-deterministic circuit families can be used here instead of advice sequences). We thus consider a real number c whose decimals encode an advice sequence for F. Given an input $u \in \{0,1\}^n$, the corresponding advice can be read (deterministically) in polynomial time by a BSS machine like in the proof of theorem 3. F can be computed in polynomial time by a *deterministic* Turing machine $\mathcal{T}$, with the help of the advice and of a (non-deterministic) guess $g \in \{0,1\}^{p(n)}$ where p is a polynomial. We have just seen that $\mathcal{T}$ can be simulated in linear time by a BSS machine. Hence it just remains to be shown how the guess can be computed by a non-deterministic BSS machine $\mathcal{M}$. This is done by the following program ($h \in \boldsymbol{R}$ is the guess of $\mathcal{M}$).

```
pow:=1;
for i:=1 to p(n) do
begin
   if h>=pow then g[i]:=1 else g[i]:=0;
   pow:=pow*2
end
```

In other words, g is the radix-2 expansion of h. If we take a guess $h \in \boldsymbol{R}^{p(n)}$, the task can be performed by an even simpler program:

```
for i:=1 to p(n) do
   if h[i]>0 then g[i]:=1 else g[i]:=0
```

□

In order to prove this result, it is also possible to simulate an off-line system, by showing that a function of PL_d can be computed in constant time, and the encoding functions of theorem 3 in polynomial time; on-line systems and Siegelmann & Sontag's neural networks are other possible starting points.

6 Final remarks

In this paper and in [7], several related models of analog computing were investigated. It turned out that in spite of several significant differences between these models, the class of polynomial-time computable functions is always P/poly. In light of this robustness property, it seems reasonable to claim that "efficient analog computing = P/poly" (compare with the *invariance thesis* of discrete complexity theory, which states that the class of polynomial-time computable functions should be the same in all "reasonable" models of computation [9]). Before a definitive conclusion can be reached, it will probably be necessary to study other models of analog computing. One such model is the real RAM machine, which was shown to be as least as powerful in section 5. We left open the important reciprocal problem of finding whether these machines can compute more than P/poly in polynomial time.

There is yet another motivation for studying this problem: an answer to this question could shed light on the "$\mathrm{P}_R = \mathrm{NP}_R$" problem proposed by Blum, Shub & Smale.

As a matter of fact, if $\mathrm{P}_R = \mathrm{NP}_R$, this equality also holds for the BSS model restricted to discrete inputs and outputs: $\mathrm{P}_{RD} = \mathrm{NP}_{RD}$. If it turns out that $P_{RD} = P$/poly, we can conclude like in [7]: since NP $\subset$ NP/poly, $\mathrm{P}_R = \mathrm{NP}_R \Rightarrow$ NP $\subset P$/poly. However, the latter inclusion violates standard hypotheses of discrete complexity theory [1]. One could thus conclude that $\mathrm{P}_R = \mathrm{NP}_R$ is very unlikely. This approach to the $\mathrm{P}_R = \mathrm{NP}_R$ problem will be developed in a forthcoming paper.

Finally we note that several new problems were recently shown to be NP_R-complete or NP_R-hard [5]. These new developments add to the interest of the $\mathrm{P}_R = \mathrm{NP}_R$ problem.

References

[1] J.L. Balcázar, J. Diáz, and J. Gabarró. *Structural Complexity I.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1988.

[2] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers : NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, July 1989.

[3] M. Cosnard, M. Garzon, and P. Koiran. Computability properties of low-dimensional dynamical systems. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, 1993. To appear in the Lecture Notes in Computer Science, Springer-Verlag.

[4] M. Cosnard and P. Koiran. Relations between models of parallel abstract machines. In *Proceedings of the Heinz Nixdorf seminar, Paderborn, November 1992.* to appear in the Lecture Notes in Computer Science, Springer-Verlag.

[5] C. Cucker and F. Roselló. On the complexity of some problems for the Blum, Shub & Smale model. In *Proceedings of Latin'92*, number 583 in Lecture Notes in Computer Science, pages 117–129. Springer-Verlag, 1992.

[6] Ker-I Ko. *Complexity Theory of Real Functions.* Birkhäuser, 1991.

[7] H. T. Siegelmann and E. D. Sontag. Neural networks with real weights: analog computational complexity. SYCON Report 92-05, Rutgers University, September 1992.

[8] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. In *Proc. Fifth ACM Workshop on Computational Learning Theory*, July 1992.

[9] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A, chapter 1. Elsevier, 1990.